Multiple Coordinated Views to Support Aspect Mining Using Program Slicing

Fernanda Madeiral Delfim and Rogério Eduardo Garcia
Departamento de Matemática e Computação
Faculdade de Ciências e Tecnologia
Universidade Estadual Paulista "Júlio de Mesquita Filho" – UNESP
Presidente Prudente - SP, Brazil
fer.madeiral@gmail.com, rogerio@fct.unesp.br

Abstract

Aspect Mining and Refactoring to Aspects aim to identify crosscutting concerns and encapsulate them in aspects, respectively. Aspect Mining remains as non-automatic process, i.e., the user needs to analyze and understand the results generated by techniques/tools, and confirm crosscutting concerns to refactor them to aspects. In this paper we propose a visual approach that deals with results generated by two aspect mining techniques proposed in the literature. By coordinating visual mappings, different levels of detail to explore software artifacts support aspect mining facilitating their interpretation for further refactoring to aspects. The model to coordinate multiple views was implemented ($SoftViz4AspectMining$ tool) and in this paper are presented the visualizations obtained, how to interpret them and lessons learned.

Keywords: Program Understanding, Aspect Mining, Refactoring to Aspects, Program Slicing, Software Visualization, Software Evolution.

1. Introduction

Modularization of *Crosscutting Concerns* (CCs) is still a challenge, since their implementations tend to be *scattered over* several system units and *tangled* with others concerns [11, 20, 14], what is a problem to understandability of software system, and as result make difficult the maintainability [4]. *Aspect-Oriented Programming* [10] provide mechanisms and abstractions to modularize CCs, encapsulating them in a separate code unit named *aspect*, increasing the modules cohesion [5] and possibility to code reuse [11]. Increasing modules cohesion makes easier program understanding activities, since it helps to focus on comprehending specific functionality implemented in separated units, supporting the software maintenance.

The evolution of existing software systems to aspect-oriented technology is also a challenge. First, it is necessary to identify CCs in non-aspect-oriented program that potentially could be turned into aspects and, then, decide about that. *Aspect Mining* is a research area aimed at CCs identification. Several techniques have been proposed to aspect mining [1, 2, 3, 8, 9, 12, 13, 22, 23]. In general, such techniques have similar limitations, like: *poor precision* of results (i.e., the percentage of relevant aspect candidates reported by a given technique is relatively low); *subjectivity* to analyze the results because the ambiguity on results produced (what on software engineer as CC candidate, other would not); *difficulty on comparing* the results of different techniques; and *difficulty on composing* of results from multiples aspect mining techniques. *Inadequate presentation of results* is pointed out as possible cause of such limitations [15]. In this context, approaches to deal with limitations of current techniques for mining aspects can be useful.

Software Visualization can be helpful on interpreting and analyzing results of aspect mining techniques, making explicit how the source code is organized, how different levels of abstraction are associated and how such associations take place in source code. Several approaches and tools have been proposed to support program comprehension using software visualization techniques, like Asbro [18], CRISTA [19] and $SoftViz_{OAH}$ [6], but they do not support CC identification.

In this paper we propose a visual approach to support aspect mining from existing object-oriented program. The proposed approach includes visualize program slicing results, allowing the visualization of control and data dependencies at statements level. The multiple coordinated views are used to allow analyzing the program structure, the associations at class level and method level based on callings, and how the slices are composed across multiple classes. We choose program slicing and added some features of the fan-in analysis in order to facilitate understanding and analyzing. Our focus is investigate whether visu-

alization contributes on understanding the results generated by those techniques (program slicing and fan-in analysis) in a complementary way. A Software Visualization tool, named $SoftViz4AspectMining$, was developed to support the visual approach proposed.

The remainder of this paper is organized as follows. In Section 2 is presented an overview about aspect mining techniques chosen. In Section 3 is presented the visual approach proposed (the coordination model) and some consideration about the developed tool. In Section 4 are presented the visual results obtained and we show how to interpret them in Section 5. Finally, the conclusions and future works are summarized in Section 6.

2. Related Works

There are several techniques proposed for mining aspects in the literature [1, 2, 3, 8, 9, 12, 13, 22, 23]. Katti et al. [9] discussed about the use of program slicing to help the process of inserting aspects in a object-oriented program. According to them, the slicing process consists of identifying the statements that form the slice (slice identification) and isolating these statements into an independent program (slice extraction). Thus, they seems very much similar to the Aspect Mining and Refactoring to Aspects process. We consider program slicing visualization promising because visual presentations can externalize slice overlapping and user interaction can be useful to deal with. Due to pages restriction, we do not present slicing techniques (see [21]).

Additionally, we also use fan-in metric mapped to visual attribute. Fan-in analysis tries to capture the scattered code at method level [16, 17]. Scattered code is a symptom of crosscuttingness because called methods might implement CCs and, then, those methods are called from many places, resulting in high fan-in value. In an aspect-oriented reimplementation of such concerns, the method would constitute an advice, and the call would correspond to the context that needs to be captured using a pointcut. The steps for fan-in analysis are: compute fan-in metric for all methods; filter relevant methods; and analyze them to determine which are CC candidates. To the last step, guidelines can be used to look for consistent invocations of methods with high fan-in value from call sites that could be captured by a pointcut definition – for example, the calls always occur at the beginning or at the end of a method [13].

3. The Coordination Model Proposed

In this paper we propose a visual approach to support aspect mining implemented in a desktop Software Visualization tool, named $SoftViz4AspectMining$. The $SoftViz4AspectMining$ architecture is organized in three layers as depicted in Figure 1.

The proposed visual approach consists in a set of visual presentations organized into *Visualization Layer* (see Figure 1) to show program features – *Aggregate View*, *Treemap View*, *Class Level Hyperbolic View*, *Method Level Hyperbolic View*, *Bars and Stripes View* and *Bytecode View*. Using a program bytecode as input, at *Data Layer* are performed static and dynamic analyzes to gather information from code, kept into data structures used to generate the visual presentations at *Visualization Layer* and to coordinate them by *Control Layer*. The multiple coordinated views enable a visual exploration of software in different levels of detail.

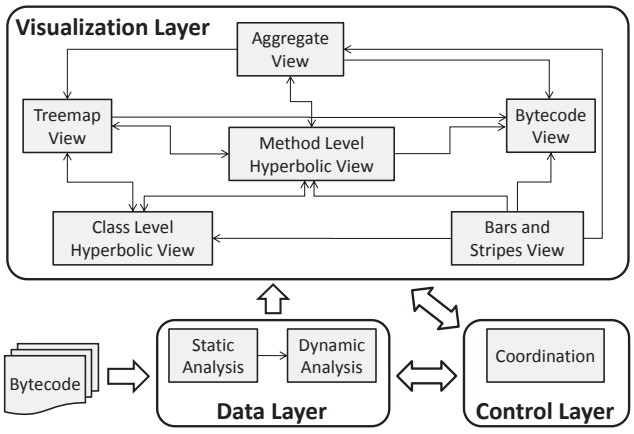

Figure 1. $SoftViz4AspectMining$ **architecture.**

The *Aggregate View* shows graphs representing instructions (vertices) linked by control flow and control and data dependency (edges) projected into regions (representing methods). Slices can be visualized and, by interacting with the *Aggregate View*, the user can choose a visual item that represents a specific statement (slice criterion) and slice type (static/dynamic and backward/forward). Then, another *Aggregate View* is created to present the slice using the parameters chosen.

The colors of visual items are predefined in each visual presentation, but the user can modify such color definition. In addition, there is a color mapping based on method calls used across several views – for each unit (i.e., packages, classes and methods) is computed the fan-in value and a color gradient from light gray to red is used to represent the obtained values (red represents the greater value). One may filter code units based on fan-in value by defining a threshold and observe them in projections and, then, it is possible to visualize units with high frequency of callings.

4. Results: Visual Presentations

The results obtained with $SoftViz4AspectMining$ are shown using a oriented-object program of an elevator simulation [7]. In Figures 2 and 3 are shown two hyperbolic projections based on method calls: the first at class level and the second at method level. The nodes represent units (classes or methods), and the direct edges represent the calling between them. The hyperbolic projection shows a node in focus (chosen by user) larger than the other and positioned in center. The neighbor nodes are colored in pink, and other nodes are no colored. Such projections help to have evidences of possible CCs, and if user decide to refactoring methods to aspect, help identify which units are affected – by analyzing the calls shown at method level it is possible to observe the method to be considered on creating advice and the ones that should be crosscut.

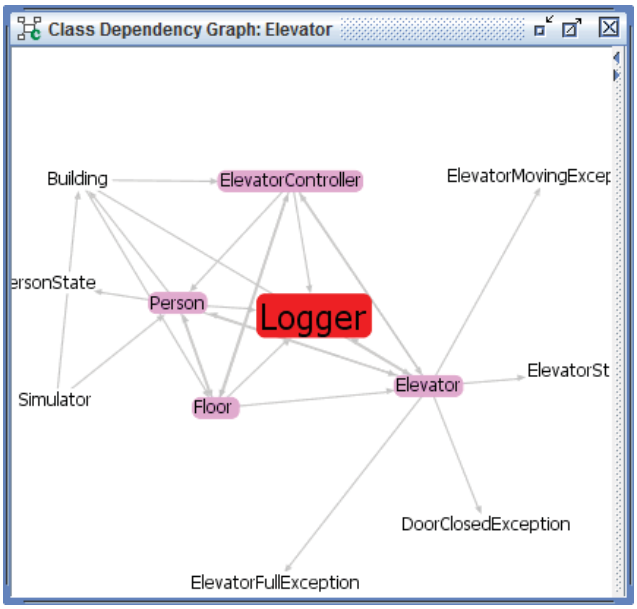

Figure 2. *Class Level Hyperbolic View*.

The *Treemap View* presents the program structure hierarchically by nested rectangles (see Figure 4). The outer rectangle represents the whole program (root). Other unit levels are packages, classes, methods and test cases, represented by inner rectangles. The rectangles are colored using a simple linear gradient of gray (dark gray represents whole program). Each leaf rectangle represents a method and its size is proportional to its amount of bytecode instruction.

In the *Aggregate View*, program instructions are represented by nodes – each node is a group of bytecode instructions. Also, the nodes are aggregated by methods, represented by curved polygon (named *aggregate region*) using specific color by method (we used transparency because of overlapping). There are two types of nodes: physical

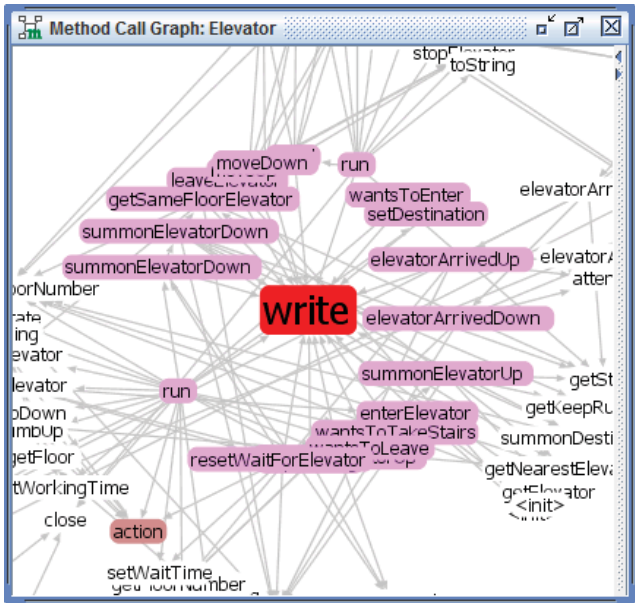

Figure 3. *Method Level Hyperbolic View*.

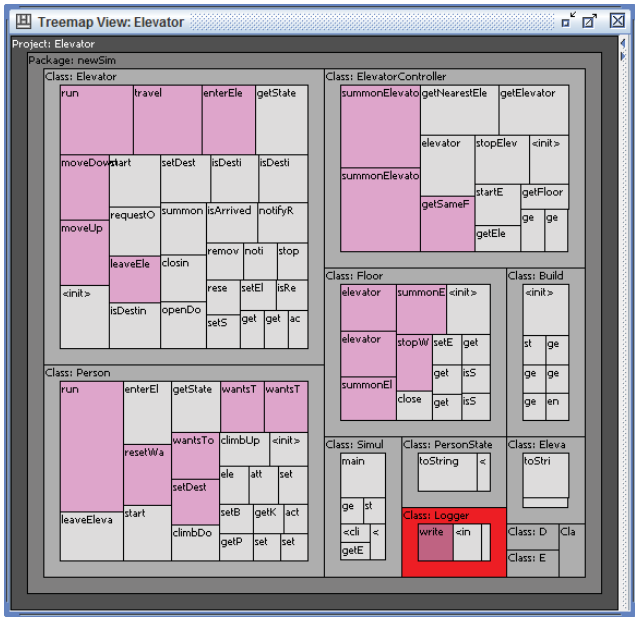

Figure 4. *Treemap View*.

nodes and virtual nodes. The physical nodes are program statements (colored in pink). The virtual nodes are introduced to represent methods entries (colored in dark green), methods exits (colored in light green) or parameters (colored in black). The edges represent control flow (continuous lines colored in black), exception (dashed lines colored in black), control dependency (continuous lines represent intra-method and dashed lines represent inter-method, both

colored in blue) or data dependency (continuous lines represent intra-method and dashed lines represent inter-method, both colored in green). *Aggregate View* allows visual exploration at statement level – statements are projected within the methods in which they are. *Aggregate Views* are generated to selected method – in Figure 5 is depicted the *Aggregate View* to method *main*. Focusing on specific node, the user can apply filters to visualize slices obtained both static/dynamic and backward/forward as subgraph.

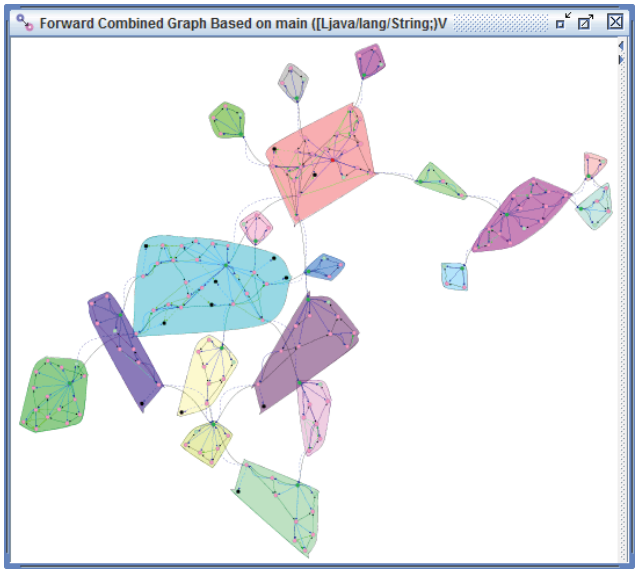

Figure 5. *Aggregate View*.

In Figure 6 is shown the *Bars and Stripes View*. The bars represent classes – the height of each bar is proportional to amount of bytecode instruction. The stripes represent program instructions using different colors assigned to each slice. Such view provides an overview of how a slice is distributed across multiple classes. Also, the $SoftViz4AspectMining$ allows visualizing bytecode instruction using two synchronized scrolling: right-scrolling to overview and left-scrolling to detail. In the right-scrolling the rectangles represent a set of bytecode instruction defined by labels in bytecode program. In the left-scrolling are shown bytecode instructions of selected rectangles in right-scrolling, grouped method which instructions belong to. Similar to *Aggregate View*, each method is shown by different color: in Figure 7 is shown the method `Logger.write (Ljava/lang/String;)V` represented in red, and other method is represented in blue. The color mapping is the same of *Aggregate View*.

5. Analyzing the Visual Presentations

In this section we show how to analyze the visual presentations. The frequency method calls is computed at level of

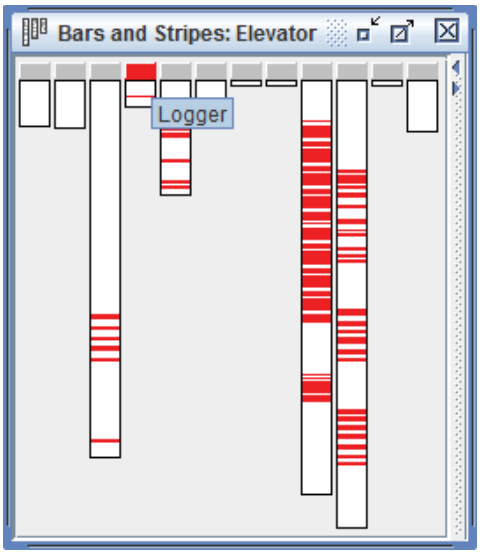

Figure 6. *Bars and Stripes View*.

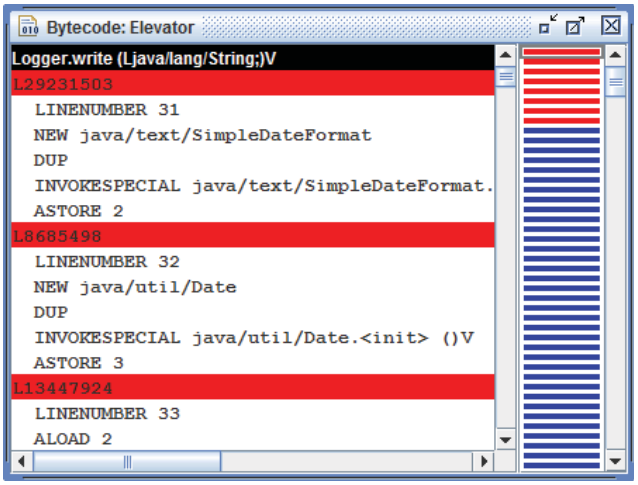

Figure 7. *Bytecode View*.

package, class/inner class/test case and method in order to create the color gradient aforementioned. In the *Class Level Hyperbolic View* (Figure 2) is shown the most called class (`Logger`) highlighted in red. The user can refine the results of class level view and methods are highlighted at *Method Level Hyperbolic View*. At method level it is also possible to use another color scale to highlight methods with high frequency call. In addition, such color scale can be used to establish a minimum threshold value to frequency call in order to discard methods. In Figure 3 are colored both `write` and `action` methods (it was considered 17 callings as minimum frequency[1]). Based on that, the user can

[1]The minimum frequency is defined empirically, taken into account the source code.

observe explicitly the frequency of calling to each filtered method and decide whether it can be considered a CC candidate.

Method Level Hyperbolic View has fundamental role in the coordination model. By selecting a method of interest, it is possible see the dependency of its class in the *Class Level Hyperbolic View* and, also it is possible to observe the selected method and the related ones at program hierarchical structure shown in *Treemap*. The *Treemap View* is not decisive for CCs identification, but it is interesting to have an overview of the program structure and observe the code units (classes and methods) that can be affected by refactoring. In Figure 4 is shown the hierarchical structure highlighting the method selected in Figure 3 (its class in red) and the related methods (called and callers) in pink. In addition, details of control and data dependency of selected method statements can be analyzed using *Aggregate View*, as well as the bytecode instructions in *Bytecode View* (see Figure 7).

The *Aggregate View* shows a graph at statements level. Each method has its own graph, and graphs can be combined by method calls statements from or to a selected method (forward or backward). A graph from a method is used to create intra-method slice, and combined graph is used to create a inter-method slice. The slices can be static or dynamic, from or to a selected node (forward or backward slice). In Figure 8 is shown a static backward slice generated from the backward combined graph based on method `write` (represents by *aggregate region* (1)). The slice criterion selected is the statement (2) and the entry node to `write` method is (3). In Figure 8 was used distance filter equal to two, from entry node (3). One may observed: 1) the statements that call directly the method owner of the slice criterion analyzing control dependency edges (blue); 2) the parameter used in the slice criterion by data dependency edges (green). Thus, the user can define pointcuts to capture joinpoints. In addition, an overview of how a slice is distributed across multiple classes is provided in Figure 6 (statements from a slice are colored in red, and the `Logger` class highlighted according color scale at class level).

By interacting with visual presentation, the user can gather information about methods and decide about refactoring to aspect or not. Methods considered CCs candidates, its class and methods callers, and how they are related must be analyzed at high and low level. At low level, the user can observe the control and data dependency, what is important to evaluate how a CC can be implemented into an aspect (for example, the aspect should have access to method data?).

In addition, we observed some considerations of program slicing. A program slicing algorithm convert the program statements into some alternative representation to calculate a slice. Since a slice consists of all the statements of a program that may affect the values of some variables in a set V at some point of interest p (slicing criterion), independently of the representation generated by program slicing algorithm, the control and data dependencies should be calculated. Thus, program slicing can be useful to help the Refactoring to Aspect process, not only to extract a slice, but also to help to define pointcuts (a step of Refactoring to Aspects).

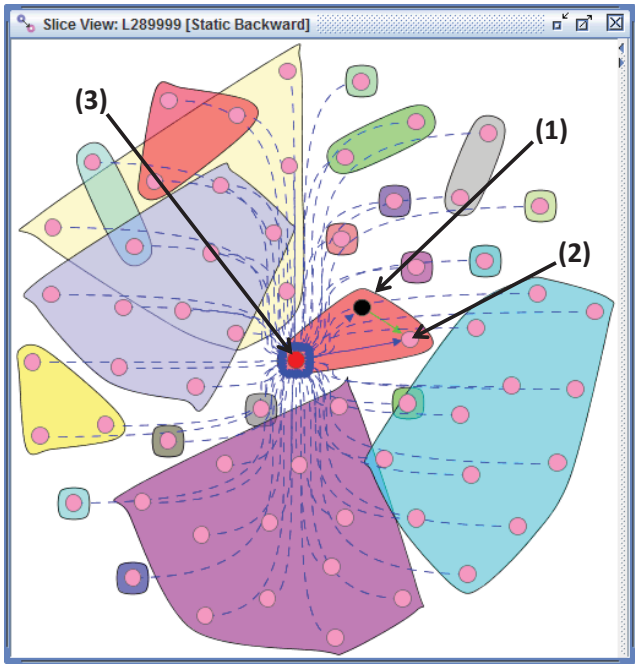

Figure 8. Slice Visualization in the *Aggregate View*.

6. Conclusions and Future Works

In this paper we propose a visual approach that consists in coordinating multiple views, using a color mapping based on fan-in metric and a slices visualization, enabling: 1) visual exploration of software in different levels of detail; 2) visualization of the methods more frequently called; 3) control and data dependency analysis to define pointcuts.

We show the $SoftViz4AspectMining$ tool that was developed to support the multiple coordinated views model. Our pilot study has shown that the model proposed supports analyzing by exploring different levels of detail, helps to deal with the limitations of projections (scalability of visualization techniques) by user interaction (applying distance filter in the graphs views, for example), and allows pointing out the visual results and observations (lessons learned) presented here. However, to evaluate the effectiveness of coordinated visual model proposed, a controlled experiment has been planned, aims to identify CCs in object-oriented program. Also, we intend to evaluate the tool focusing on

defining pointcuts (how effective is the help to refactor to aspects). The initial study was very important to identify possible treats to controlled experiment.

In addition, we intend to implement other aspect mining techniques, to combine them in a complementary way. For example, *link analysis* [8] identifies CCs at class level, and may be advantageous make explicit the analysis result on *Class Level Hyperbolic View*, as well as the result of *clone detection* in the *Aggregate View*, with slices information.

References

[1] E. S. Abait, S. A. Vidal, and C. A. Marcos. Dynamic Analysis and Association Rules for Aspects Identification. In *Proceedings of the II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP '08)*, pages 31–39, October 2008.

[2] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *19th IEEE International Conference on Automated Software Engineering (ASE '04)*, pages 310–315, Washington, DC, USA, September 2004. IEEE Computer Society.

[3] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[4] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect Mining Techniques. *Software Quality Control*, 14(3):209–231, September 2006.

[5] C. Clifton, G. T. Leavens, and J. Noble. MAO: Ownership and Effects for More Effective Reasoning About Aspects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '07)*, pages 451–475. Springer-Verlag, 2007.

[6] A. F. d'Arce, R. E. Garcia, R. C. M. Correia, and D. M. Eler. Coordination Model to Support Visualization of Aspect-Oriented Programs. In *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE '12)*, pages 168–173, 2012.

[7] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[8] J. Huang, Y. Lu, and J. Yang. Aspect Mining Using Link Analysis. In *Proceedings of the 5th International Conference on Frontier of Computer Science and Technology (FCST '10)*, pages 312–317, Washington, DC, USA, 2010. IEEE Computer Society.

[9] A. Katti, V. Bingi, and V. Chavan. Application of Program Slicing for Aspect Mining and Extraction - A Discussion. *International Journal of Computer Applications*, 38(4):12–15, January 2012.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, pages 327–354, London, UK, UK, 2001. Springer-Verlag.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, pages 220–242. Springer-Verlag, 1997.

[12] J. Krinke and S. Breu. Aspect Mining Based on Control-Flow. In *Proceedings of the 7th Workshop Software Reengineering (WSR '05)*, volume 25, pages 39–40, Bad Honnef, Germany, May 2005. GI-Softwaretechnik-Trends.

[13] M. Marin, A. V. Deursen, and L. Moonen. Identifying Crosscutting Concerns Using Fan-In Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(1):3:1–3:37, December 2007.

[14] P. Massicotte, L. Badri, and M. Badri. Towards a Tool Supporting Integration Testing of Aspect-Oriented Programs. *Journal of Object Technology*, 6(1):67–89, 2007.

[15] K. Mens, A. Kellens, and J. Krinke. Pitfalls in Aspect Mining. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 113–122, Washington, DC, USA, October 2008. IEEE Computer Society.

[16] A. Mubarak, S. Counsell, and R. M. Hierons. An Evolutionary Study of Fan-in and Fan-out Metrics in OSS. In P. Loucopoulos and J.-L. Cavarero, editors, *Proceedings of the 4th IEEE International Conference on Research Challenges in Information Science (RCIS '10)*, pages 473–482. IEEE, May 2010.

[17] E. Nasseri, S. Counsell, and E. Tempero. An Empirical Study of Fan-In and Fan-Out in Java OSS. In *Proceedings of the 8th International Conference on Software Engineering Research, Management and Applications*, pages 36–41, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[18] J.-H. Pfeiffer and J. R. Gurd. Visualisation-Based Tool Support for the Development of Aspect-Oriented Programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD '06)*, pages 146–157, New York, NY, USA, 2006. ACM.

[19] D. Porto, M. G. Mendonça, and S. C. P. F. Fabbri. CRISTA: A Tool to Support Code Comprehension Based on Visualization and Reading Technique. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '09)*, pages 285–286, 2009.

[20] C. N. Sant'Anna, A. F. Garcia, C. v. F. G. Chavez, C. J. P. de Lucena, and A. v. Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

[21] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[22] P. Tonella and M. Ceccato. Aspect Mining Through the Formal Concept Analysis of Execution Traces. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society.

[23] T. Tourwé and K. Mens. Mining Aspectual Views Using Formal Concept Analysis. In *Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM '04)*, pages 97–106, Washington, DC, USA, 2004. IEEE Computer Society.